# 艦これAPIを叩く

masarakki

Published
with GitBook

# Table of Contents


1. Introduction
2. まえがき
3. ログイン
4. API概略
5. データ取得API
6. 編成・補給・修理・遠征
7. 出撃・戦闘
8. まとめ
9. あとがき

# C86同人誌 艦これAPIを叩く

## Build

```
bundle install
rake
```

`c86.pdf` will be generated!

wishlistはこちら

# まえがき

APIにアクセスすることを一般的に「APIを叩く」と表現します。 本書は艦これのAPIを叩き、どのようにゲームが進行しているか解説します。

なんてのは嘘です。 この本のタイトルの「艦これのAPIを叩く」はアクセスするという意味ではありません。

世間一般で「XXを叩く」といえば、「XXを非難する」という意味の比喩表現です。 この本も、人類史上稀に見る「クソAPI」である艦これのAPIを「非難」し、 艦これAPIを悪いAPIの題材とし、よりよいAPI設計を解説することを目的としています。

なお、著者は2013年の10月初旬に**BAN**されて、 抜け道[1]も2014年2月初旬に閉じられてしまい、

現在は既にプレイできないので、書かれたAPIがそのまま実行できる保証はありません。

特に4月頃に大幅にAPIが変わったという話があるので、 今は少しは改善されているのかも知れません。

本書の内容は *2014*年*1*月末日 時点のAPIを元に書かれています。 艦これAPIやDMMゲームの仕様に関しては、通信内容やhtmlやjavascriptを読んで推測したものであり、 内部の動作が実際にそうであるということは保証できません。 なお、過去のAPIに対応した自動化システム(以下 `IJN48` )を**Github**で公開しています。

> 1. BANされたのに抜け道があって半年閉じられなかったというクソAPI ↩

# ログイン

この章では艦これにログインするまでのやりとりを解説します。

# DMMにアクセスしてアプリを起動する

ゲームを始めるために、DMMにアクセスして艦これを起動します。

DMMのオンラインゲームは、他の多くのSNS上のソーシャルゲームと同様に、 DMMが用意する共通レイアウトの中に、ゲームプロバイダから指定された `gadget.xml` というファイルの内容に従って、 `iframe` でゲーム本体を読み込む仕組みになっています。

DMMの内部では、アプリのキーやシークレットと共に、 `gadget.xml` へのURLが登録されていると推測されます。 アプリ起動に必要な情報は全てDMMが持っているので、簡単に攻撃はできません。

## アプリ起動中に攻撃できる人

艦これの `gadget.xml` は、httpで配信されています。 `gadget.xml` にアクセスするクライアントは**DMM**サーバなので、httpでも外部からの攻撃には耐性があります。 ただし、DMMのサーバと艦これのサーバを繋ぐ経路上にいる機器を攻撃できた場合、 ニセの `gadget.xml` を配信することができるので**Man in the Middle** 攻撃(以下**MitM**攻撃)が成立します。 プロバイダの中の人とかじゃないとできないと思います。

# 艦これサーバにユーザ認証

`iframe` に艦これ本体が読み込まれると、艦これサーバへログイン認証が行われます。 まずjavascriptでDMMサーバにアクセスしているユーザの情報を取りに行きます。 DMMのユーザIDが取得できると、艦これサーバに接続し、所属する鎮守府を取得し、ログインAPIを叩きます。

ログインAPIが成功すると、 レスポンスには**api_token**と**api_starttime**という2つの情報が含まれていて、 この2つの情報をパラメータにしてflashを読み込み、ゲームが始まります。 ちなみに `IJN48` では、このflashを読み込む瞬間を**Chrome Extension**でフック)し、 ローカルに立てたサーバにトークンを送信して、認証情報を外の世界に送る[1]という工夫をしています。

ユーザがBANされている場合、ログインAPIは特殊なレスポンスを返し、 クライアントは@{banned}のような画面を表示します。




## トークンの仕組み

後述するAPIは全て、この**api_token**と**api_starttime**を利用してリクエストします。 この**api_token**から艦これサーバでユーザを特定し、 **api_starttime**で正しいリクエストか検証しているようです。 逆に言うとこのパラメータさえ知っていれば、他のユーザに成りすますことが可能です。 このパラメータは艦これのユーザ認証を行うたびに新しい値が払い出されます。 イメージとしてOAuthのアクセストークンに近いものがあります。

## ユーザ認証中に攻撃できる人

ほぼすべての人が攻撃できます。 この認証システムにはたくさんの問題があります。

### HTTP通信

サーバ間通信ではそれほど問題になりませんが、 今回はjavascriptによりユーザのブラウザと艦これサーバがログインの通信を行います。 MitM攻撃でログインの通信を覗かれた場合、 攻撃者は簡単にトークンを取得できるので何でもやり放題です。

ユーザとサーバ間のMitM攻撃は圧倒的に簡単に実現可能です。 パケットキャプチャを仕込んだ**WiFi**アクセスポイントを設置すればいいだけです。 外でも艦これするためにWindowsタブレットがバカ売れらしいですが、 パスワードがかかってない野良**WiFi**を見つけて、しめしめなんて思って繋いでしまうと、 通信内容が全て覗かれている可能性があります。 その場合 `api_token` が抜かれてしまうのであらゆるAPIが叩かれてしまいます。

## 生存期間

どうやら艦これのユーザ認証を行って新しいトークンを取得しない限り、トークンは無限に生き続けるようです。 BANされていても約半年間ゲームができたのは、このトークンを自分のコンピュータのデータベースが覚えていたからで、覚えていたトークンをパラメータにして**flash**を直接リクエストすれば問題なくゲームができました。 BANが完全に有効に

なった理由は明らかではないですが、少なくとも半年の生存期間があるようでした。

### BANの判定タイミング

BANの判定はログイン時のみ行われ、新しいトークンが取得できないという制限しかありません。 以前に取得した生きているトークンさえ知っていればBANが有効になりません。 逆に新しいトークンが取得できないことで、生きているトークンが特定しやすいと言っても過言ではないです。 全ての**API**アクセス時にサーバでユーザのBAN判定をするべきです。 どうせトークンからユーザを特定する作業はしているはずなので負荷は増えません。

## まとめ: API改善ガイド

- 当たり前だけど全部**HTTPS**にする
- トークンの生存期間を十分短い時間に設定する
- リクエスト毎にトークンを作りなおす/取得し直す
- リクエスト毎にサーバ側でBANの判定をする
- キーが2つあるのはそんなに意味がない `api_starttime` は不要

特にトークンを毎回作りなおすのは非常に効果があります。 APIレスポンスに次のトークンを含めるとかでいいと思います。 攻撃者が一度トークンを取得したらずっと覗き続けられることが現状の実装の一番マズい部分でしょう。 毎回トークンが変われば、攻撃者が1回APIレスポンスを見逃しただけで続きが見れなくなります。 他の効果として、 `IJN48` を動かしながら普通にゲームをプレイするのがかなり難しくなります。

ただしAPIリクエストの並列化ができなくなるので大きな問題になります。 現実的な解としては、トークンの生存期間を十分短くし、頻繁にトークン更新をさせることでしょう。

> 1. この部分が一番独創的な部分だと思ってます ↩

# API概略

この章では艦これのAPIのリクエストとレスポンスの共通部分を解説します。

# リクエスト

## プロトコル

全て `http` です。 あらゆるものが丸見えです。

`MitM` で認証情報を取得した攻撃者は、全てのAPIを実行可能です。

### HTTP メソッド

全てのAPIリクエストが `POST` メソッドを使っています。 `POST` メソッドは、リソースの作成に使われるべきメソッドで、 例えば艦娘のデータを取得するというAPIは、 艦娘というリソースに変化が起こらないので、 `GET` を使うべきですが、 全て `POST` が使われています。

## 認証

ログインの際に渡された `api_token` をパラメータに追加してユーザ特定を行います。 実はこれだけではなく、 `Referer` ヘッダーにflashのurlを設定する必要があり、 そのurlには `api_token` と `api_starttime` のパラメータが付けられています。

## その他

`Origin` ヘッダーにAPIサーバのドメイン、 `Accept-Encoding` に `gzip, deflate, sdch` の指定が必須でした。

# レスポンス

## 形式

レスポンスは全て**JSON**です。 正確に言うと、

```
svdata={JSON文字列}
```

という、 `eval` で評価すると変数 `svdata` に JSON のデータが格納される `javascript` のコードです。 たぶん `svdata` はグローバル変数で、APIのレスポンスに対する処理は1つのグローバル変数を共有しているのではないかと思います。

## 文字コード

文字コードは**UTF-8**で統一されていますが、APIのモノによってはなぜか**BOM**付き**UTF-8**になっていました。 レスポンスを出力してもBOMは目に見えないので、JSONを取得するための `s/^svdata=//` が上手く動かない理由に気づきにくく、非常に困りました。もしわざとやっているならなかなか優秀な罠です。

ソースコードの保存時に起きた問題と考えられ、開発者のエディタやその設定が共通化されていないことが想像できます。 また、ファイルの先頭バイトがレスポンスにそのまま出力されている[1]という特徴から、 サーバの言語は**php**ではないか[2]と予想しています。

## ステータスコード

APIレスポンスのステータスコードは全て `200 OK` です。 `200 OK` はリクエストが成功したことを表すステータスコードであり、 不正なリクエストや内部エラーなどでリクエストが正しく処理できない場合は、 `200 OK` ではなくふさわしいステータスコードを返すべきです。

ステータスコードの代わりに、JSON 内に `api_result` という情報があり、この値が `1` の場合はAPIは成功したとみなされます。 不正なリクエストや内部エラーなどでリクエストが成功しなかった場合、 `api_result` は `1` 以外の値になり、このときflashクライアントは猫画面を表示します。

===[column] ねこのひみつ APIリクエストの失敗理由が(たぶん)高負荷の場合などは、 同じリクエストを続けて送ると普通に成功することがあります。 しかしflashクライアントはそれを一切することなく、ただ闇雲にリロードしろと迫ってくるのです。 艦これ最大の敵と言われる妖怪猫吊るしはこのような仕組みで提督の精神を削っていくのです。

### JSONデータ

APIレスポンスの主要なデータは、 `api_data` というキーの中に入っています。 大雑把に書くとこのようなデータになっています。 Github には大量のレスポンスのサンプルがあります

```
{"api_result": 1,
 "api_data": {
    "api_count": 15,
    "api_page_count": 3,
    "api_list": [
       {"...."},
       {"...."},
       ...
    ], ...
  }
}
```

なぜか全てのキーに `api_` という接頭詞が付けられており非常にデータサイズが無駄です。 レスポンスのデータサイズはそのまま通信量のサイズになり、 現代のコンピュータ環境では通信スピードが一番のボトルネックなので、 すなわちどれくらいのリクエストを処理できるかという話になります。 試しにいくつかのレスポンスから `api_` の文字を抜いてみたら、データサイズが約**2**割削減できました。 単純に `api_` を抜くだけで、通信コストや同時アクセス数、flashの負荷が約2割ほど改善されるということです。

## まとめ: **API**改善ガイド

- HTTPSを使う
- HTTPメソッドとステータスコードを正しく使う
- BOM付きUTF-8 や `svdata=` はコードからお里が知れるから削除する
- データ削減を意識してレスポンスの構造を最適化する

Google先生が1バイトを削るのに心血を注いでいるように、 レスポンスを1バイト削ることは積み重なって大きな改善になります。 個別のレスポンスを見ていくと、酷いデータ構造もたくさんあり、 データ量の削減を意識してデータ構造を再設計するとかなり負荷は改善されると思います。

> [1]. <?php の前にBOMのバイト列が書かれていることになる ↩

> [2]. テンプレートに紛れ込んでいる可能性もあるがJSONを個別にテンプレートファイルに書くとは思えない ↩

# データ取得API

この章では艦これを構成する様々なデータがどのように配信されるのか解説します。

# 2種類のデータ

データには大きく分けて2つの種類があります。 マスタデータとユーザデータです。

マスタデータは基本的にログイン時にまとめて取得し、 ユーザデータは行動ごとに必要なものを取得します。

ログイン時にロードで長時間待たされるのは、大量のマスタデータを取得するためだといえます。

# マスタデータ

マスタデータのAPIは全て `/kcsapi/api_get_master/` というURLの下にあります。 ログイン時に艦種、艦娘データ、装備、家具 などの固定データをひと通り取得します。 例外はマップに関するマスタデータで、これは出撃のたびに取得されます。

例えば艦種データを見てみるとid=1が海防艦だったり、戦艦が2種類あったり、 なかなか不思議なものを見た気になります。

## 艦娘マスタデータ

艦娘のマスタデータは、約50KBある巨大なデータです。 このデータにはやはり不要なデータがたくさん入っています。 初期のデータはなるべく小さく、必要になった時点で他のデータを取ってくる設計にするとかなり小さくなるはずです。

例えば、全員分のドロップ時のメッセージ[1]が含まれています。 回数的に考えると、戦闘終了時のAPIレスポンスに毎回含めたほうが全体のサイズは小さくなると思います。

他にも図鑑で使うためか能力の初期値が全員分含まれていますが、これも図鑑にアクセスした瞬間に取得するべきです。 一回のログインで一度も図鑑にアクセスしない人のほうが多分多いんだから。

# ユーザデータ

ユーザデータのAPIは全て `/kcsapi/api_get_member/` というURLの下にあります。 所持している艦娘、装備、家具や艦隊の情報、レベルや資材の数などが取得できます。

## 艦娘データ

所持している艦娘が全て詰まったデータです。 1ページに表示されるのはたかだか20件くらいだったと記憶していますが、それでも全件取得します。 これを提督の部屋に戻るたびに毎回取得しに行きます。

艦隊編成画面のソート順をグローバルに持っていて、その順でソートされたレスポンスが返ってきます。 艦隊編成画面でソート順を変更すると、全部取得しなおしです。

それぞれの艦のデータは


```
{
  "api_member_id":32377,
  "api_id":2,
  "api_sortno":347,
  "api_name":"\u6dbc\u98a8\u6539",
  "api_yomi":"\u3059\u305a\u304b\u305c",
  "api_stype":2,
```

```
    "api_ship_id":247,
    "api_lv":77,"api_exp":349951,"api_afterlv":0,
    "api_aftershipid":0,
    "api_nowhp":15,"api_maxhp":30,
    "api_taik":[30,57],"api_souk":[49,49],"api_houg":[53,49], ...
  }
```

のようなデータになっていて、これが全員分配列で並んだデータになっています。

見て解るとおり、 `ship_id` が指定されているにも関わらず、 艦娘のマスタデータを一切使わずすべて冗長なデータになっています。 この中で必要なデータは**id, ship_id,** 現在のステータス値だけであることは容易に解ります。 ソート順の変更も、ソートされた**id**のリストでよいはずです。

### 武器データ

もっと凶悪なのが武器データです。 所持している武器を全て取得します。 艦娘の数とは比べ物になりません。 データサイズでおよそ6倍[2]ありました。 武器を破棄が凄く遅いのは、破棄後に毎回このAPIが呼ばれるためです。

艦娘と同じように全て冗長なデータになっています。 武器は成長の要素がないので、**id**何番の武器を何個持っているというレスポンスで十分です。 たぶん現在のレスポンスを9割くらい削減できるのではないでしょうか。

## 初期化処理の最適化サンプル

初期化処理を最適化してみましょう。 まずユーザデータのうち、レベルや資材数は現状のAPIのままでよいでしょう。

ログイン直後にマスタデータを取得する必要はありません。 最短で提督部屋の画面を出すために、秘書官の**ID**と内装の**ID**リストだけを返すAPIを作りましょう。 秘書艦のログイン時ボイスや、内装の画像は**ID**から**url**が決まるようにルールづけしましょう[3]。 これだけで提督部屋の画面が出せます。たった数キロバイトで実現可能です。

その後おいおいマスタデータや他のユーザデータを取得するようにしましょう。

## まとめ: API改善ガイド

- とにかくデータ量を減らす
- データの分割・差分取得をする
- 冗長なデータを避け正規化する
- 非同期通信を活用する

中身のコードがわからないので正確なところはわかりませんが、 レスポンスの `svdata=` はおそらくグローバル変数への代入のような処理だと想像します。 つまり、APIのレスポンスを受け取るとグローバル変数が書き換えられることになり、 これを上手く制御して動かすには、通信が逐次実行であることが保証されてないといけません[4]。

「複数のリクエストを同時に出し、返ってきたレスポンスから処理する」という非同期通信の強みが使えないことになります。 さらに、レスポンスを見て新しくリクエストを作ることも難しいので、 「必要なデータがなかったら差分取得」みたいなコードも書きにくいのではないでしょうか。

> 1. 例) 艦隊のアイドル、那珂ちゃんだよー。よっろしくぅ〜! ↩

> 2. 所持艦娘: 30KB 所持装備: 200KB ↩

> 3. たぶん最初からそうなってる ↩

> 4. 逐次実行は理解しやすいので悪いわけではない 特にゲームの進行などでは 適材適所である ↩

# 編成・補給・修理・遠征

この章では通常画面でする艦隊の操作に関するAPIを解説します。

## 編成

艦隊の編成を変更するには、 `/kcsapi/api_req_hensei/change` のAPIを叩きます。 パラメータは、 第 `api_id` 艦隊の `api_ship_idx` 番目を `api_ship_id` の艦娘に変更するという形式です。 艦娘を外す場合には `api_ship_id` に `-1` を指定します。

Flashクライアントからは1つの艦隊に同じ種類の艦娘を配置できないようになっていますが、 APIでは何も制限されておらず、同じ種類どころか全く同じship_idで6隻埋めることも可能です。 今のところ伊401はイベント報酬の1隻しか無いはずですが、全員伊401で埋めることが可能ということです。

さらに、通常変更できない状況でもAPIでは何も制限しません。 出撃中[1]の艦隊でも遠征中の艦隊でも変更可能です。

## 補給

艦隊に補給するには、 `/kcsapi/api_req_hokyu/charge` のAPIを叩きます。 `api_id_items` に補給する艦娘のidを `,` で繋げた文字列を指定します。

編成と同様に制限はほとんど無く、 出撃中の補給も遠征中の補給も自由です。 Flashでは1ページごとに補給しなくてはいけませんが、APIはおそらく全艦一気に補給できます[2]。

## 修理

艦娘を修理するには、 `/kcsapi/api_req_nyukyo/start` のAPIを叩きます。 パラメータには `api_dock_id` と `api_ship_id` 、 バケツを使う場合は `api_highspeed` を `1` に設定します。 途中でバケツを使う場合は、 `/kcsapi/api_req_nyukyo/speedcharge` のAPIを叩きます。

もう予想できると思いますが、修理のAPIにも制限はほとんどなく、 出撃中でも遠征中でも修理できます。 ボスに到達したら全艦娘にバケツぶっかけてなんてこともできてしまいます。 唯一チェックが入るのはドックが使用済みかどうかというところだけです。

## 問題点

API設計の基本中の基本です。必ずサーバ側で入力が正しいかチェックしましょう。 APIのみならず、webシステムを作るときは必ずサーバ側で入力値のチェックをしましょう。

例えばhtml5には入力値のチェック機構が入りましたが、あれは単なる入力補助でしかありません。 javascriptで入力値のチェックをしたりすることもありますが、 それも余計なHTTPリクエストを減らしたいという目的で使うことがほとんどです。 その場合にも必ずサーバ側で入力値のチェックをしましょう。

## 根本的な問題

そもそも、サーバ側で入力値のチェックをしていないということは、 エンジニアがある致命的な状態に陥っていることを示しています。 それはクライアントプログラムを信用しているという間違った思想です。

Webシステムを作る上で、絶対にクライアントを信用してはいけません。 HTTPの仕様上、何が正しいのか、何が正しくないのか判断することはできません。 そもそも正・不正の概念を定義することすら不可能なんじゃないでしょうか。 全ての通信は、実行するユーザには基本的に全て見えてしまうし、 それを完全再現することも容易です。 なので絶対にクライ

アントを信用してはいけません。

実際、 `IJN48` を実装する時も通信の内容を解析しかしていません。 やっていることもflashクライアントが出すリクエストと同じものを出しているだけです。 flashクライアントの逆アセンブルは多分規約で禁止されている[3]とおもいますが、そんなもの全く意味がないのです。

さて、 `IJN48` とは何でしょうか? **API**を叩く**bot**でしょうか? それとも**flash**のプログラムを実行する処理系でしょうか? それともブラウザそのものでしょうか? 運営の言う「正しくプレイしないとBANするぞ」の正しいプレイとは何でしょうか? 本質的にそんなもの定義できないんですよ。 ネットワーク上では全ての通信が平等です。 そこには正しいとか正しくないとか無いのです。

===[column] セキュリティ設計のはなし

Webシステムの設計でクライアントを信用してはいけないように、 セキュリティの設計を考える時は、絶対に人間を信用してはいけません。 基本的に人間は全員犯罪者で常に攻撃の機会を狙っていると思わなければいけません。 この「人間」には、 同僚や親兄弟はおろか自分自身も含みます。 エンジニアは日々、システムを知り尽くした自分ならどんな犯罪を犯すかを想像し、 それでも攻撃できないよう、ありとあらゆる可能性を潰す、という仕事をしています。

そりゃ病むに決まってんだろ・・・

# まとめ: API改善ガイド

- サーバで入力値を全てチェックする
- クライアントを信用しない
- 正しいプレイなんか定義できない

とはいうものの、正しいプレイを定義する方法は実は1つだけあります。 それは指定された環境以外での実行を許可しないという方針にすることです。 艦これの場合、OSとそのバージョン、ブラウザとそのバージョン[4]、 flashプレイヤーの製造元とバージョンを指定して許可すればよいでしょう。 もちろんこれはクソ不自由で**Web**の世界の理念に反することは明らかですし、 とりあえず**Opera**ユーザが発狂する姿がありありと目に浮かびます[5]。

> 1. 出撃中の操作はたしか冬イベントあたりに大幅に修正されて今はもうできない ↩

> 2. そんな恐ろしいことやったことありません ↩

> 3. 昔からほとんどのソフトウェアの契約条項に逆アセ禁止が含まれている ↩

> 4. chrome拡張で同じことをするまでです 拡張が入ったchromeでのプレイは禁止するって!? ハハ、こやつめ! ↩

> 5. Linuxユーザが発狂する姿も見えますね・・・ ↩

# 出撃・戦闘

この章ではゲームの華である戦闘のAPIを解説します。

# 出撃・戦闘の流れ

戦闘に関係するAPIは

- 移動系API
- 戦闘系API
- 結果系API

の3種類に分類できます。 これを繰り返して戦闘します。

## 移動系API

出撃するマップを選んだ時の「出撃API」と、戦闘後に進軍を選んだ時の「進軍API」がなぜか同じレスポンスです。 羅針盤が回っている時には既に行き先は決定しています。

レスポンスには移動先の情報が含まれていますが、本当に知性を感じられない設計になっています。 特に重要な戦闘の属性は `api_color_no` という意味の分からないキー名が振られています。 夜戦スタートとかは判明したんですが潜水艦の判定を出してるのがどこなのかは最後まで判明しませんでした。

あと物資がもらえるマスを通っても物資が増えないという現象に遭遇しました。 これも最後まで何をすれば回収できるのか判明しませんでした[1]。

## 戦闘系API

移動先の戦闘属性に従って適切な戦闘APIを叩く必要があります。 戦闘APIは「通常戦闘」「夜間戦闘」「夜から昼」の3種類があります。 「通常戦闘からの夜戦」と「夜間戦闘」は別のAPIです。

戦闘APIを叩くと戦闘結果が返ってきます。 信じがたいほど理解しにくいJSONで、知性の欠如を感じます。

例えば砲撃の結果は、 `api_hougeki` の中に `api_def_list` と `api_damage` の2つの配列[2]が入っています。

```
{
  "api_hougeki": {
    "api_def_list": [10, 3, 12,  1, ...],
    "api_damage":   [25, 3, 15, -1, ...],
    ...
  }
}
```

このデータを、 `api_def_list` の `n` 番目に書かれた艦[3]に `api_damage` の `n` 番目に書かれた数字をダメージとして与える、というルールで処理していきます。

`api_hougeki` を配列に、その中の要素をハッシュにしたほうが圧倒的に理解しやすい表現です。

```
{
  "api_hougeki": [
    {"attacker":  1, "defender": 10, "damage": 25},
    {"attacker": 11, "defender":  3, "damage":  3},
    {"attacker":  4, "defender": 12, "damage": 15},
    {"attacker": 12, "defender":  1, "damage": -1},
    ...
  ]
}
```

サイズが気になるなら、可読性は落ちますがハッシュではなく配列にしてもいいでしょう。

```
{
  "api_hougeki": [
    [ 1, 10, 25],
    [11,  3,  3],
    [ 4, 12, 15],
    [12,  1, -1],
    ...
  ]
}
```

この場合コードによる補足が無いと理解しづらいですが、 `api_hougeki` が配列なだけで遥かにマシです。 APIレスポンスは人間の理解できる構造を表現するべきであり、 元のレスポンスは `for` 文 というコンピュータの処理の都合に合わせて作られていて不適切です[4]。

なお、去年の冬くらいに、 「戦闘APIを叩いたあとは10秒間待たないと次のAPIリクエストがことごとくエラーになる」ように仕様変更されました。 この変更により、ただでさえ面倒な戦闘が致命的に面倒になったので、 イベント以外のプレイを完全に辞めました。

### 結果系API

結果APIを叩くとそのマスでの戦闘が終了します。 夜戦に行かない場合も結果APIを叩きます。 レスポンスは関心がないので調べてませんが、経験値とかドロップとかが書かれているんだと思います。

マップの終端に到達した場合や進撃せずに撤退する場合は、母港画面に戻り、艦隊情報の取得などの一連APIが叩かれます。 この一連のAPIの中にある `/kcsapi/api_auth_member/logincheck` というAPIが、 実は出撃状態だったら出撃を完了するという処理をしています。 まったく名が体を表していないし、ユーザからしたら完全に副作用です。いみわかんない。 出撃中に猫った場合、再ログインさせられてれて母港に飛ぶので出撃状態が解除されます。

## 問題点

戦闘の流れはとにかく理解しにくいです。 完全に場当たり的な実装を繰り返しているのが想像できます。 レスポンスも整理されておらず、違うAPIなのに同じようなレスポンスを返していたりします。 出撃という行動のモデリングができてないと思われます。

### RESTという考え方

RESTでは、物事をリソースとしてURLで表現し、HTTPメソッドを正しく使ってそのリソースを操作します。 RESTとAPIは相性がよく、よい設計の助けになります。

# RESTful戦闘API設計サンプル

出撃-戦闘-結果 の一連のAPIをRESTfulにしてみましょう。 戦闘に限らずゲームはリソース作成/操作の積み重ねです。 全てのリソースを頭から再評価すれば最後の状態を復元できるようにします。

### 出撃

まず1つ目のリソースは出撃です。英語だと**sotie**です。 まず出撃するには `POST` メソッドで新しい `sotie` のリソースを作成します。 `map_id` や `fleet_id` などがパラメータになるでしょう。

```
POST /soties
```

レスポンスはこのようになります。

```
{
  "id": 2341234,
  "fleet_id": 1,
  "area_id": 3,
  "map_id": 2,
  "done": false
}
```

すでに出撃中の場合[5]、APIは `400 Bad Request` を返して失敗します。

この出撃のリソースには何時でも、何回でもアクセスでき、常に同じ情報が取得できます[6]。

```
GET /soties/2341234
```

出撃を終了するには `DELETE` メソッドを使います。

```
DELETE /soties/2341234
```

リクエストを見るだけで `done` が `true` に設定されると想像できるでしょう[7]。

## 移動

移動もリソースと考えて良いと思います。 リソース名は**move**です。 リソースは名詞というルールがあるので、動詞のmoveではなく名詞のmoveであることを忘れてはいけません。 `POST` メソッドで移動を作成します。

```
POST /soties/2341234/moves
```

移動できる場合は、移動先の情報と次にすべきことを示したデータが取得できます。

例えば戦闘が必要なマスに移動した場合はこのような感じでしょうか。

```
{
  "id": 98765,
  "cell": 1,
  "type": "battle",
  "submarine": true,
  "boss": false,
  "next" "battle"
}
```

みんな大好き物資マスはこんな感じですね。

```
{
  "id": 98766,
  "cell": 2,
  "type": "supply",
  "items": {
    "steel": 15,
    "fuel": 5
  },
  "next": "move"
}
```

移動できない場合[8]は、 `400 Bad Request` を返します。 レスポンスの中身に戦闘しろ、とか帰還しろ、のように書いてもいいと思います。

もちろん作ったリソースは `GET` で何度でも情報は取得できるし、 更に全ての移動経路も取得できるAPIも欲しいですね。

```
GET /soties/2341234/moves/98765
GET /soties/2341234/moves
```

## 戦闘と結果

戦闘するときも、まず `POST` メソッドで**battle**を作ります。

```
POST /soties/2341234/battles
```

元のAPIだと通常戦闘なのか、夜戦なのか、などによって呼び出すAPIが違いましたが、 戦闘というリソースは一つのはずなので分けるべきではありません。

このようなレスポンスを返しましょう。

```
{
  "id": 132435,
  "type": "normal"
}
```

とりあえず戦闘のリソースを作りました。 このとりあえずというのが重要です。 サーバは現在の状態を知っているので、 `sotie_id` と `cell` が自動で記録されます。 APIレスポンスに載せるかどうかは別の話です。

通常戦闘の場合、陣形を選択します。 これは、作った戦闘のリソースの陣形を変更すると考えます。 存在するリソースへの変更は `PUT` を使います。

```
PUT /soties/2341234/battles/132435

formation=3
```

これは何度でも実行できるべきです。 また、変更できない場合[9]は、 `400 Bad Request` を返してもいいし、 `200 Success` を返しつつ変更は無視してもいいでしょう。

戦闘結果を取得しましょう。

```
GET /soties/2341234/battles/132435/result
```

もちろん `GET` なので何度叩いても良いし、常に同じ結果を返します。 最初にリクエストされた時に戦闘の計算を行い、戦闘結果を確定し、データベース等に記録します。 艦隊へのダメージや、燃料砲弾の減算の反映なども行います。 内部的に持っている昼戦確定フラグのようなものも立てておきます。 いったん昼戦確定フラグが立ったら、陣形の変更は二度とできないようにしなければなりません。 2度目以降のアクセスは、データベース等に記録された内容を返します。

通常戦闘はこのあと夜戦に行くことができます。 やはりこれも戦闘リソースを変更すると考えます。

```
PUT /soties/2341234/battles/132435

night_battle=true
```

夜戦した場合の戦闘結果も同様です。

```
GET /soties/2341234/battles/132435/result
```

夜戦結果が無かったら計算を行い、記録し、結果を返します。 戦闘リソースとしては1つなので、昼戦の結果に更に夜戦の結果が追加されたような形になります。 夜戦だけの結果を返すのは間違っています。 もちろん夜戦確定フラグのようなものを立てておきます。 もう `night_battle=false` にすることはできません。

戦闘が終わったら経験値と艦娘ゲットの時間です。 `PUT` で終了を伝えましょう。

```
PUT /soties/2341234/battles/132435
complete=true
```

その後

```
GET /soties/2341234/battles/132435/result
```

を叩くと、経験値やドロップが追加された戦闘結果が取得できます。

ゲームでは順々に処理されていくので無理ですが、APIではこのようにリクエストすることも可能です。

```
PUT /soties/2341234/battles/132435

formation=3&night_battle=false&complete=true
```

これでresultを叩けば一発で結果が取得できます。

コンピュータの都合でJSONの構造を決めるのも間違っていますが、 ゲームの都合でAPIを実装するのも間違ったことです。

### 状態の復旧

現在のAPIでは、何かエラーがあるとすぐに猫画面を出し、 とにかくリロードさせ、全てを諦めて母港に飛ばします。 この仕様は本物の糞です。

最後にどの状態だったのか、サーバは知っているはずです。 きちんと設計すれば直前の状況まで復旧することは可能です[10]。 今回の実装例だと、過去も含めて全ての出撃を再現可能です[11]。

この点パズドラは本当に上手くやっていると思います。

### 問題点

たぶんこの戦闘APIの設計には色々反論したい人がいると思います。 夜戦は新しいリソースとして作るべきだとか、それぞれのresult分けるべきだとか、 `POST /battles` の瞬間に結果を確定させるべきだとか、まぁいろいろ思うところはあると思います。 これもただの一例なので、よく考えて作ろうとか俺はこんなふうに思考して作ってるって部分が伝わればいいと思います[12]。

このAPI設計の問題点として、明らかに更新の処理が複雑になるのが目に見えています。 どのパラメータを弄れるかというのがかなり状態に依存するので、 さらに内部を綺麗に設計しないと巨大な `if` 文の闇になるでしょう。 ただし他の部分に闇がなくなるので、 このような闇を一箇所に押し込めた設計は完全な悪では無いと思います。

## まとめ: API改善ガイド

- リクエストとレスポンスを1対1対応させる
- JSONは処理の都合ではなくあるべき構造に合わせる
- リソースの概念を理解しAPIを整理する
- 状態での分岐を減らす

根本的な問題は、度重なる仕様変更でどんどん戦闘システムが複雑になっていることです。 その対応として場当たり的な拡張を続け、はっきり言ってシステムは破綻しています。 艦これが圧倒的につまらない原因はこの複雑な戦闘システムです。

外から見ただけでは分からないのでここでは触れていませんが、 実際の戦闘の処理も相当複雑なコードになっていると思います。 特にイベントボスの弱点に特定の装備を設定しているところなど、 API設計の能力を見る感じではたぶん相当複雑でカオスなコードになっているんじゃないでしょうか。 あと3回くらいイベントをやった後くらいには、 戦闘の処理をする関数を含むファイルは社内で『聖典』などと呼ばれるようになると予測します。 そうなるともう手が付けられず、誰も変更の影響範囲を正しく把握できず、 少しの変更でもバグが出て実装に何週間もかかるような状態になります。

まずはとにかくゲームシステムの単純さを取り戻すことが何より重要だと思います。

> 1. オリョクル? なにそれ美味しいの? ↩

> 2. 誰が攻撃したかは戦闘結果に関係ないので無視している ↩

> 3. 敵味方艦混合でインデックスが振られる 味方艦:1-6 敵艦:7-12 みたいな感じに ↩

> 4. 元レスポンスの配列は固定長 つまり固定回数の `for` を回している可能性がある 圧倒的にダサい ↩

> 5. つまり `done` が `false` な `sotie` が存在する場合 ↩

> 6. このような性質を冪等性と言う HTTPの `GET` は冪等性を持つべきとされている ↩

> 7. この方法を論理削除という 記録を残す必要がなければ単に物理削除してもよい ↩

> 8. 戦闘しないといけない場合や終端の場合など ↩

> 9. 少数編成の場合や通常戦闘でない場合 ↩

> 10. 現状でも可能だと思うんですけどね・・・出撃終了条件を変な扱いしてるから・・・ ↩

> 11. 出撃時の艦隊の状態を保存する必要があります ↩

> 12. こわいこわいRESTfulおじさんに見つかりませんように・・・ ↩

# まとめ

API設計に大事なことのまとめです。

# セキュリティ

ありとあらゆる攻撃を考慮しないといけません。

ユーザの環境も様々です。 APIを提供する側はそれらを制限することなんてできません。 クライアントに正しいも間違っているもありません。 不正な攻撃であろうと全てのリクエストは平等に扱われてしまうことを忘れてはいけません。

とにかく想像力が必要です。 人間は全員犯罪者で、ありとあらゆる手段を使って攻撃をしてきます。 どんな攻撃ができそうか想像し続けましょう。 単純にシステム上のやりとりだけではありません。 ソーシャルハックもかなり有効な攻撃手段です。 (ハッカーに奪われた、5万ドルのTwitterアカウント「@N」:オーナーに返還 見事なソーシャルハック事例) 他のサービスの情報と組み合わせた攻撃[1]も有効です。 本当に穴はないですか? XSSのように機械的に検出できるようなものではありません。 複雑なシステムの一つ一つは小さな穴でも、組み合わせたら大きな穴になることもあります。 本当に穴はないですか?

とりあえずAPIの提供は必ずHTTPSを使いましょう。 経路上でおこるほとんどの攻撃はこれで防げます[2]。 今どきはAPIだけでなく全てのWebアクセスをHTTPSにするべきです。 Google大先生に倣いましょう。

APIレスポンスの内容には十分注意をはらいましょう。 余計な情報から実装が透けて見える場合があります。 他人が知ってはいけない情報を絶対に流してはいけません。 その情報を他人が知った場合どのような攻撃に使えるか[3]熟考して、必要な情報のみを選びましょう。 「自分のじゃない場合の個人情報はクライアントアプリで無視する」みたいな実装を提案してくるバカがいます。 死ねばいいと思います。

そして何より、不要な個人情報を知ろうとしないのがもっとも効果的です。 OAuthを使えばパスワードなんか知る必要はありません。 パスワードがなければリスト型攻撃も怖くありません。 住所や電話番号を知る必要があるサービスなんてごく一部です。 知る必要のない情報は知るべきではない、というスタンスを貫くと身軽になります。

===[column] ソーシャルハック 秘密なように見えてパブリックな情報はたくさんあります。 例えば、秘密の質問によくありがちな「母親の旧姓は?」なんて家族や親戚にはノーガードです。 もちろん家族や親戚も犯罪者であることを忘れてはいけません。 田舎の場合近所のほとんどの人が知っていることもあるでしょう。 「通っていた小学校の名前は?」も全然秘密じゃないです。Facebookに書いてあります。 「飼ってるペットの名前」は積極的に出していってますよね。 「2段階認証で誕生日を入力してください」なんて完全にZALですねぇ。

人間は生きているだけで大量の情報を垂れ流しています。 垂れ流したほうがWebは面白いので積極的に垂れ流したほうがいいと思います。 個人に本物の秘密など無いとセキュリティ設計者は考えたほうがいいでしょう。 いろいろな公然の秘密と、サポートのような人間相手の交渉が組み合わさると、 システムのセキュリティは意外なほど脆く崩れ去ります。

# RESTful

世の中猫も杓子もRESTfulなのでRESTfulにしましょう。 この業界、長いものに巻かれるのは大正義です。

モノをリソースとして捉え、その操作をHTTPメソッドに割り当てます。 制約を与えることで設計に統一感が出て、表現力が高まります。 当然ですがHTTPメソッドを正しく使いましょう。 GETは必ず冪等性を保証しなければいけません。

HTTPステータスコードも正しく使いましょう。 失敗しているのに `200` で返してはいけません。 JSONの中の `success` で判断しろなんてAPIはクソです。

最初はリソースの設計も難しいでしょう。 とことん考え、議論することが重要です。 仕事では1つのURIを決めるのに3日議論したこともあります。 それだけ名前をつけるというのは重要な行為です。

ついでに認証の方法はOAuthにしましょう。 艦これの場合、OAuth + 余計な検証になっています。 意味のない検証をやめ

て素直なAPIアクセスが可能であるべきです。 リファラ検証などはやってはいけません。 gzipが使えないクライアントにもきちんと対応しましょう。 世の中のすべてのブラウザを把握するなんて無理です。

## JSON

大前提として、APIレスポンスとしてのJSONは人間が読むものです。 人間がもっとも理解しやすいデータ構造にするべきです。 ぱっと見ただけで意味がわかる構造になってますか? コンピュータでの処理を先に考えてデータ構造を作るとだいたい間違ったことになります。 世界を正しく認識しデータ構造をモデリングしましょう。

すべてのキー名を熟考しましょう。 全く初見でも意味が理解できるキー名になってますか? きちんとした英単語を使いましょう。 絶対に単語を省略してはいけません。 複数のものならきちんと複数形を、 状態を表す場合は形容詞がふさわしいかそれとも過去分詞か[4]よく考えましょう。 英語力はかなり必要です。 というかエンジニアに一番必要なものは間違いなく英語力なので英語の勉強は必ずしましょう。

ただデータサイズが大きくなりすぎると、通信コストとしてジワジワ効いてきます。 可読性と通信コストはトレードオフな部分もあるので[5]、 本当に必要であればサイズに最適化する場合もあります。

# まとめのまとめ

3つに共通することはとにかく考えることです。 最近は言語やフレームワークの発達で実装に掛かる時間は大きく短縮しています。 その分とことん深く長く考えることができます[6]。

機能が正しく動くのは当たり前のことです。 今の時代、動けば良いなんて考えは通用しません。 理解しやすいか、保守しやすいか、拡張しやすいか、 ひっくるめて如何によい設計か、というところが求められます。 どれだけアタマと時間を使ったかがAPI設計の良し悪しが決まります。 考え尽くして正しい設計を導き出すのがエンジニアの最大の使命です。

## コスト

最後にすべての判断基準となるコストの話です。 システムのコスト源は

```
人間 > ネットワーク > (壁) > ストレージ > (超えられない壁) > CPU
```

であることを忘れてはいけません。 今は無限にサーバが増やせる時代なので、サーバを増やせば済むことはサーバを増やすべきです。 ストレージもちょっと高いけど無限です。 それに比べてネットワーク帯域は料金も高く、なにより通信速度に限界[7]があり、大きなデータを送受信することはユーザ環境に依存するストレス源でもあります。

人間は最大のコスト源ですが、ここをケチると糞システムになり最終的なコストが増大します。 高コストであると同時に、最も価値を発揮する装置です。 人間はコンピュータにはできない仕事をしないといけません。 設計などの考える仕事がそれです。 実際にコードを実装する部分も、未だに機械は完全には[8]こなしてくれません。 良い設計から良い実装をするのもやはりまだ人間でなくてはできない仕事です。

人間がやるべき仕事をエンジニアが最高のパフォーマンスでできるということを最優先しましょう。

> 1. 他のサービスから流出したパスワードで不正アクセスする最近大人気のリスト型攻撃 ↩

> 2. HeartBleed... うッ 頭がッ... ↩

> 3. 最近の小学生は登下校中は名札を外すよう教えられる 名前を知られた場合どのような攻撃を受けるか考えてみよう ↩

> 4. 俺もよくわかってない ↩

> 5. 計算の処理コストは比較にならないほど小さい ↩

> [6]. もし今でも実装に手一杯というなら言語やフレームワークの選択が間違っているのか もしくは才能がないのでエンジニア辞めたほうがいいです ↩

> [7]. 一般的な家庭向け回線速度は10年前と比較しても理論値で2倍程度にしかなっていない ↩

> [8]. フレームワークはずいぶん発達し大規模なコードの自動生成を行うが完全な全自動化はまだまだ先の話だろう ↩

# あとがき

前回は落選だったので1年ぶりのサークル参加です。 いかがだったでしょうか。 冬コミ時期はtwitter絶賛大炎上中だったので、まあ落ちてよかったのかも。

# 今回の本について

今回の本も面倒な人に見つかったら炎上の可能性高いですね。 今まで作った本の中で一番攻めている感じだと思います。 今までぬるぬるやっていたのに一転攻勢です。 健全な人生を送ってきた自称良識派の人は表紙を見ただけで全力で非難してくるでしょうね。

とはいえ、この本のきっかけは「俺だったら艦これのクソAPIをどう直すか」という話であり、 こういうクズなAPIを実装してしまう人を1人でも減らすことを目的にしています。 APIの具体的な内容を深く解説しなかったのはこういう理由なのです。 至って真面目で非常に教育的であり健全ですね。

この本を書きながら技術的な検証をしている際に、 本当に踏んではいけない脆弱性を見つけてしまい、ゴニョゴニョゴニョしました。 本当に冷や汗出ました。 今はまだ話せないのでいずれ話せる機会があったら笑い話にしたいです。

# Re:VIEW

今回の執筆フォーマットはRe:VIEWを使ってみました。 実際にはmd2reviewを使ってMarkdownからRe:VIEWに変換してます。 基本はMarkdownの文法で書いて、 Re:VIEW特有のタグはそのままMarkdownに書くと問題なく変換できてものすごく便利です。

ただ自由度はだいぶ低く未実装な機能も多いようなので苦労します。 原稿部分は楽なんだけどあとがきと奥付を追加するのに非常に苦労しました。 というか今からRe:VIEWのコードを弄って欲しいアウトプットを実装します・・・

っと上の文章を書いてから2時間ほどRe:VIEWのコードと格闘していたんですが、 なんですかこのクソコードは・・・ これはマジでちょっとビビるレベルのクソコードですよ。 夏コミが無事に終わったら冬に向けてRe:VIEWにプルリク送りまくるしかないと思いました。

# 今後について

そろそろ別のこともしてみたいなぁと思ってみるけどコード書く以外何もできねえんだなぁ俺。 そろそろ Crash of Clansのストレスが高くて自動化したいと思っているので、上手くいけたらそのネタやるかもしれません。 あと久しぶりにエロ漫画の原作書きたいです。